

# In-Parameter-Order: A Test Generation Strategy for Pairwise Testing

Jeff Lei

Department of Computer Science and Engineering

The Univ. of Texas at Arlington

6/21/2005

# Outline

- Introduction
- The IPO Strategy
- Related Work
- 3-Way Testing and Beyond
- Conclusion

## Why Testing?

- ❑ Modern society is increasingly dependent on the quality of software systems.
- ❑ Software failure can cause severe consequences, including loss of human life
- ❑ **Testing** is the most widely used approach to ensuring software quality

# The Testing Process

The testing process consists of three stages:

- **Test Generation** - Generate test data inputs
- **Test Execution** - Test setup and the actual test runs
- **Test Results Evaluation** - Check if the output is in line with expectations

## The Challenge

- ❑ Testing is labor intensive and can be very costly
  - often estimated to consume more than 50% of the development cost
- ❑ Exhaustive testing is impractical due to resource constraints
- ❑ How to make a good trade-off between test effort and quality assurance?

## Pairwise Testing

- ❑ Given any pair of input parameters of a system, every combination of valid values of the two parameters be covered by at least one test
- ❑ A special case of **combinatorial testing** that requires **n**-way combinations be tested
  - **n** can be 1, 2, ..., or the total number of parameters in the system
- ❑ Based on simple specifications, and does not need to look into the implementation details

## Example (1)

Component			
Web Browser	Operating System	Connection Type	Printer Setting
Netscape	Windows	LAN	Local
IE	Macintosh	PPP	Networked
Mozilla	Linux	ISDN	Screen

TABLE I

FOUR COMPONENTS, EACH WITH THREE SETTINGS

Exhaustive testing requires 81 tests =  $3 * 3 * 3 * 3$ .

## Example (2)

<b>Test</b>	<b>Browser</b>	<b>OS</b>	<b>Connection</b>	<b>Printer</b>
1	NetScape	Windows	LAN	Local
2	NetScape	Linux	ISDN	Networked
3	NetScape	Macintosh	PPP	Screen
4	IE	Windows	ISDN	Screen
5	IE	Macintosh	LAN	Networked
6	IE	Linux	PPP	Local
7	Mozilla	Windows	PPP	Networked
8	Mozilla	Linux	LAN	Screen
9	Mozilla	Macintosh	ISDN	Local

TABLE II

TEST SUITE TO COVER ALL PAIRS FROM TABLE I

## Why Pairwise?

- ❑ Many faults are caused by the **interactions** between two parameters
  - 92% **block** coverage, 85% **decision** coverage, 49% **p-uses** and 72% **c-uses**
  
- ❑ Not practical to cover all the parameter interactions
  - Consider a system with **n** parameter, each with **m** values. How many interactions to be covered?
  
- ❑ A “good” trade-off between test effort and test coverage
  - For a system with 20 parameters each with 15 values, **pairwise testing** only requires less than 412 tests, whereas **exhaustive testing** requires  $15^{20}$  tests.

# Outline

- ❑ Introduction
- ❑ The IPO Strategy
- ❑ Related Work
- ❑ 3-Way Testing and Beyond
- ❑ Conclusion

## NP-Completeness

- The problem of generating a **minimum** pairwise test set is NP-complete.
  - Can be reduced to the **vertex cover** problem
- Unlikely to find a **polynomial** time algorithm to solve the problem.
  - **Greedy algorithms** are the first thing coming into the mind of a computer scientist

# The Framework

## Strategy In-Parameter-Order

begin

/\* for the first two parameters  $p_1$  and  $p_2$  \*/

$T := \{(v_1, v_2) \mid v_1 \text{ and } v_2 \text{ are values of } p_1 \text{ and } p_2, \text{ respectively}\}$

if  $n = 2$  then stop;

/\* for the remaining parameters \*/

for parameter  $p_i, i = 3, 4, \dots, n$  do

begin

/\* horizontal growth \*/

for each test  $(v_1, v_2, \dots, v_{i-1})$  in  $T$  do

replace it with  $(v_1, v_2, \dots, v_{i-1}, v_i)$ , where  $v_i$  is a value of  $p_i$

/\* vertical growth \*/

while  $T$  does not cover all pairs between  $p_i$  and

each of  $p_1, p_2, \dots, p_{i-1}$  do

add a new test for  $p_1, p_2, \dots, p_i$  to  $T$ ;

end

end

# Horizontal Growth

**Algorithm** *IPO\_H*( $\mathcal{T}, p_i$ )

//  $\mathcal{T}$  is a test set. But  $\mathcal{T}$  is also treated as a list with elements in arbitrary order.

{ assume that the domain of  $p_i$  contains values  $v_1, v_2, \dots$ , and  $v_q$ ;

$\pi = \{ \text{pairs between values of } p_i \text{ and values of } p_1, p_2, \dots, \text{ and } p_{i-1} \}$ ;

if ( $|\mathcal{T}| \leq q$ )

{ for  $1 \leq j \leq |\mathcal{T}|$ , extend the  $j$ th test in  $\mathcal{T}$  by adding value  $v_j$  and  
remove from  $\pi$  pairs covered by the extended test;

}

else

{ for  $1 \leq j \leq q$ , extend the  $j$ th test in  $\mathcal{T}$  by adding value  $v_j$  and  
remove from  $\pi$  pairs covered by the extended test;

for  $q < j \leq |\mathcal{T}|$ , extend the  $j$ th test in  $\mathcal{T}$  by adding one value of  $p_i$   
such that the resulting test covers the most number of pairs in  $\pi$ , and  
remove from  $\pi$  pairs covered by the extended test;

}

}

# Vertical Growth

```
Algorithm IPO.V( $\mathcal{T}, \pi$ )  
{ let  $\mathcal{T}'$  be an empty set;  
  for each pair in  $\pi$   
  { assume that the pair contains value  $w$  of  $p_k$ ,  $1 \leq k < i$ , and value  $u$  of  $p_i$ ;  
    if ( $\mathcal{T}'$  contains a test with “-” as the value of  $p_k$  and  $u$  as the value of  $p_i$ )  
      modify this test by replacing the “-” with  $w$ ;  
    else  
      add a new test to  $\mathcal{T}'$  that has  $w$  as the value of  $p_k$ ,  $u$  as the value of  $p_i$ ,  
      and “-” as the value of every other parameter;  
  };  
   $\mathcal{T} = \mathcal{T} \cup \mathcal{T}'$ ;  
};
```

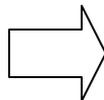
## Example (1)

Consider a system with the following parameters and values:

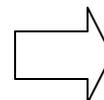
- parameter **A** has values **A1** and **A2**
- parameter **B** has values **B1** and **B2**, and
- parameter **C** has values **C1**, **C2**, and **C3**

## Example (2)

<u>A</u>	<u>B</u>
A1	B1
A1	B2
A2	B1
A2	B2



<u>A</u>	<u>B</u>	<u>C</u>
A1	B1	C1
A1	B2	C2
A2	B1	C3
A2	B2	C1



<u>A</u>	<u>B</u>	<u>C</u>
A1	B1	C1
A1	B2	C2
A2	B1	C3
A2	B2	C1
A2	B1	C2
A1	B2	C3

Horizontal Growth

Vertical Growth

## PairTest

- ❑ A Java tool that implements the IPO strategy
- ❑ Supports the following types of test generation
  - Account for **relations** and **constraints**
  - Extend from an existing test set
  - Modify/extend an existing test set after changes of **parameters, values, relations** and **constraints**
- ❑ Has been used in IBM and software engineering classes at NCSU

# Empirical Results (1)

Let  $n$  be the number of parameters, and  $d$  the domain size of each parameter. The size of a pairwise test set is in the order of  $O(\log n)$  and  $O(d^2)$ .

Results of PairTest for Systems with  $n$  4-Value Parameters

$n$ (# of parameters)	10	20	30	40	50	60	70	80	90	100
$s$ (# of tests)	31	34	41	42	48	48	51	51	51	53
$t$ (time in seconds)	0.11	0.16	0.22	0.44	0.77	0.99	1.37	1.81	2.23	2.96

Results of PairTest for Systems with 10 Parameters,  
Each Having  $d$  Values

$d$ (# of values)	5	10	15	20	25	30
$s$ (# of Tests)	47	169	361	618	956	1355
$t$ (time in seconds)	0.05	0.28	0.72	1.54	2.96	5.16

## Empirical Results (2)

Sizes of Pairwise Test Sets Generated by AETG and PairTest

System	S1	S2	S3	S4	S5	S6
AETG	11	17	35	25	12	193
PairTest	9	17	34	26	15	212

S1: 4 3-value parameters

S2: 13 3-value parameters

S3: 61 parameters (15 4-value parameters, 17 3-value parameters, 29 2-value parameters)

S4: 75 parameters (1 4-value parameter, 39 3-value parameters, 35 2-value parameters)

S5: 100 2-value parameters

S6: 20 10-value parameters

# Outline

- ❑ Introduction
- ❑ The IPO Strategy
- ❑ Related Work
- ❑ 3-Way Testing and Beyond
- ❑ Conclusion

# Classification

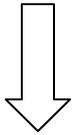
- **Computational** methods that are mainly developed by computer scientists
  - **AETG** (from Telcordia), **TCG** (from JPL/NASA), **DDA** (from ASU), **PairTest**
  
- **Algebraic** methods that are mainly developed by mathematicians
  - **Orthogonal Arrays**
  - **Recursive Construction**

## AETG (1)

- ❑ Starts with an empty set and adds one (complete) test at a time
- ❑ Each test is **locally optimized** to cover the most number of missing pairs:
  - Generate a random order of the parameters
  - Use a greedy algorithm to construct a test that covers the most uncovered pairs
  - Repeat the above two steps for a given number of times (suggested 50), and select the best one

# AETG (2)

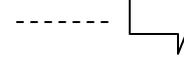
<u>A</u>	<u>B</u>	<u>C</u>
----------	----------	----------



<u>A</u>	<u>B</u>	<u>C</u>
A1	B1	C1



<u>A</u>	<u>B</u>	<u>C</u>
A1	B1	C1
A1	B2	C2



<u>A</u>	<u>B</u>	<u>C</u>
A1	B1	C1
A1	B2	C2
A2	B1	C3
A2	B2	C1
A2	B1	C2
A1	B2	C3

Adds the 1st test

Adds the 2nd test

Adds the last test

## AETG vs IPO

- ❑ AETG is fundamentally **non-deterministic**, whereas IPO is **deterministic**
- ❑ AETG has a higher order of complexity, both in terms of time and space, than IPO
- ❑ AETG is a commercial tool, and its license is very expensive, whereas IPO is open to the public.

## Orthogonal Arrays (1)

- An orthogonal array  $OA_\lambda(N; k, v, t)$  is an  $N \times k$  array on  $v$  symbols such that every  $N \times t$  sub-array contains all tuples of size  $t$  from  $v$  symbols *exactly*  $\lambda$  times.
  - $N$  - Number of test cases
  - $k$  - Number of parameters
  - $v$  - Number of values of each parameter
  - $t$  - Degree of interaction
  - $\lambda$  - 1 for software testing and is often omitted
  
- For example, Table 2 is an orthogonal array  $OA(9; 4, 3, 2)$

## Orthogonal Arrays (2)

OA (9; 4, 3, 2)

$(b_0, b_1)$	$A = b_1$	$B = b_0 + b_1$	$C = b_0 + 2 * b_1$	$D = b_0$
(0, 0)	0	0	0	0
(0, 1)	1	1	2	0
(0, 2)	2	2	1	0
(1, 0)	0	1	1	1
(1, 1)	1	2	0	1
(1, 2)	2	0	2	1
(2, 0)	0	2	2	2
(2, 1)	1	0	1	2
(2, 2)	2	1	0	2

## Orthogonal Arrays (3)

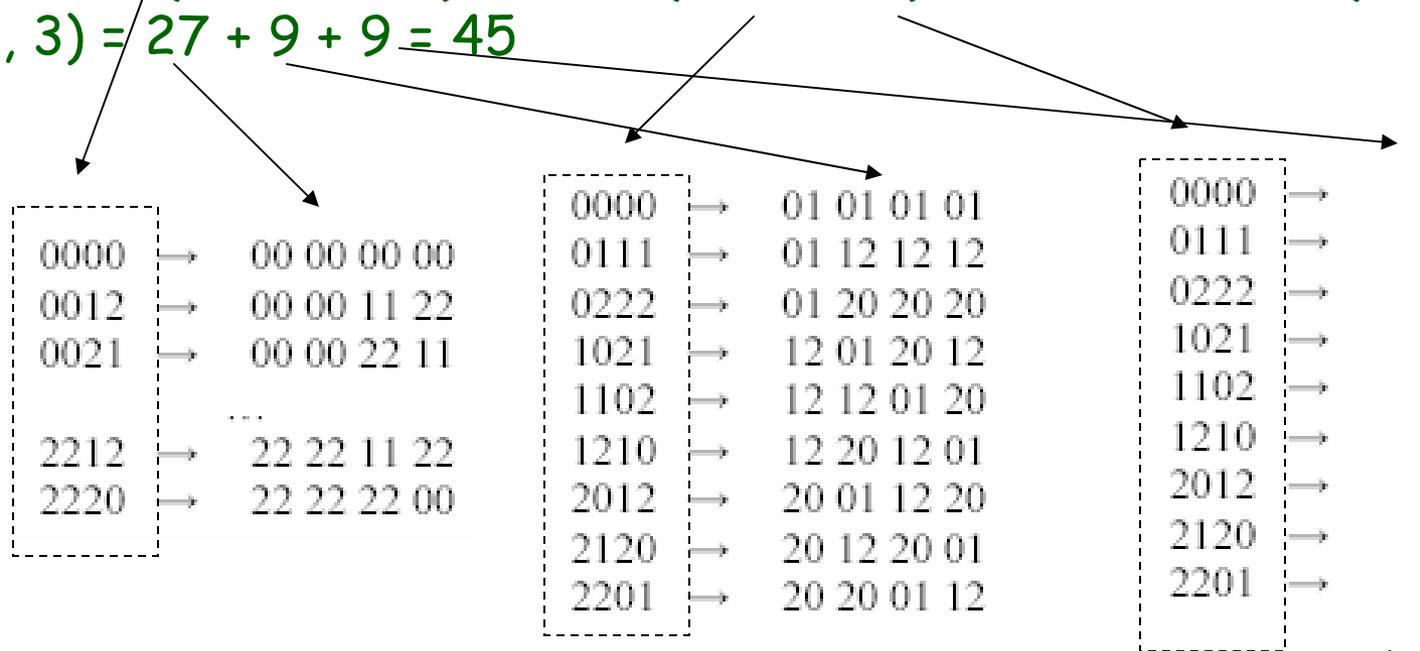
- **Orthogonal arrays** can be constructed very fast and are always optimal
  - Any extra test will cause a pair to be covered for more than once
  
- However, there are several limitations:
  - **Orthogonal arrays** do not always exist
  - Existing methods often require  $|v|$  be a prime power and  $k$  be less than  $|v| + 1$ .
  - Every parameter must have the **same** number of values
  - Every  $t$ -way interaction must be covered at the **same** number of times

## Recursive Construction (1)

- ❑ **Covering arrays** are a more general structure, which requires every **t-way** interaction be covered **at least once**
- ❑ Constructing a **covering array** from one or more **covering arrays** with smaller parameter sets
- ❑ **Recursive construction** can be fast, but it also has restrictions on the number of parameters and the domain sizes

# Recursive Construction (2)

Use  $OA(27; 4, 3, 3)$  and  $OA(9; 4, 3, 2)$  to construct  $CA(27; 8, 3, 3) = 27 + 9 + 9 = 45$



↑  
Double each column

↑  
0 → 01  
1 → 12  
2 → 20

↑  
0 → 02  
1 → 10  
2 → 21

# Outline

- ❑ Introduction
- ❑ The IPO Strategy
- ❑ Related Work
- ❑ 3-Way Testing and Beyond
- ❑ Conclusion

## Why beyond 2-way?

- ❑ Software failures may be caused by more than two parameters
  - A recent NIST study by Rick Kuhn indicates that failures can be triggered by interactions up to 6 parameters
  
- ❑ Increased coverage leads to a higher level of confidence
  - Safety-critical applications have very strict requirements on test coverage

# The Challenges

- ❑ The number of tests may increase rapidly as **the degree of interactions** increases
  - Assume that each parameter has 10 values. Then, pairwise testing requires at least 100 tests, 3-way testing at least  $10^3$  tests, 4-way testing at least  $10^4$  tests.
- ❑ Test generation algorithms must be more sensitive in terms of both time and space requirements
- ❑ The need for **test automation** becomes even more serious
  - Impractical to manually execute and inspect the results of a large number of test runs

## State-of-the-Art

- ❑ Both **algebraic** and **computational** methods can be extended to 3-way testing and beyond
- ❑ However, **algebraic** methods have fundamental restrictions on the systems they can apply.
- ❑ **Computational** methods are more flexible, but none of them are optimized for **n**-way testing with  **$n > 2$** .

## Opportunities (1)

- Possible ideas to reduce the number of tests
  - **Domain partitioning** - identify equivalence values of each parameter
  - **Parameter constraints** - exclude combinations that are not meaningful from the domain semantics
  - **Fault-oriented test generation** - only include combinations that may contribute to one or more specific classes of faults
  - **Test budget** - maximize the coverage of n-way interactions within a given number of tests

## Opportunities (2)

- Possible ways to improve the test generation algorithms
  - Combination of **algebraic** and **computational** methods,
    - e.g., **computational methods** can be used to compute a starter covering array and then **recursive construction** can be used to expand the array

## Opportunities (3)

- Possible ideas for test automation
  - **Test harness** that can automate test setup, test execution, and test results evaluation
  - Automatically generate **test oracles** from a high level specification or by integration with tools based on formal methods, e.g., model checkers

# Outline

- Introduction
- The IPO Strategy
- Related Work
- 3-Way Testing and Beyond
- Conclusion

## Conclusion

- ❑ The problem of **combinatorial testing** is well-defined and has been used widely in practice.
- ❑ The IPO strategy is **deterministic**, has a **lower** order of complexity, and still produces competitive results.
- ❑ Algebraic methods, if applicable, are fast and can be optimal, whereas computational methods are heuristic but very flexible.
- ❑ Going beyond 2-way testing presents challenges and opportunities to the area of **combinatorial testing**.